中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

《Rust语言与内存安全设计》

第5讲 内存管理、安全性、面向对象的Rust

课程负责人：陈文清 助理教授

chenwq95@mail.sysu.edu.cn

2023年09月27日

插播一些信息

Rust语言在中国大学内普及状况调研报告【2022】

发布于 2022-08-31 10:33:59 628 0 举报

在 Rust 语言发布七周年（2022.5.15）之际，本人代表Rust 中文社区发起本次调研活动，旨在了解 Rust 语言在广大高校中的教育普及状况。本次调查报告一共有224人参与，其中有效数据为219份。希望这份报告可以反映出广大学生对于学习 Rust 语言的渴望和呼声。

参与统计的大学

> “排名不分先后，非常感谢大家的参与！

有 Rust 相关课程的学校：

清华大学、北京大学、西北工业大学、电子科技大学、中山大学

有 Rust 相关实践的学校：

上海交大实验室、上海大学、上海应用科技大学

参与调查但未开设 Rust 相关课程实践的大学：

北京邮电大学、华东师范大学、杭州电子科技大学、江苏科技大学、长江大学、西安电子科技大学、青岛科技大学、山东大学、 贵州大学、西南科技大学、安徽工业大学、电子科技大学、重庆理工大学、大连民族大学、东莞理工学院、西安电子科技大学、福州大学、国防科技大学、华中科技大学、沈阳理工、广州大学、中央民族大学、重庆邮电大学、河南科技大学、江西师范大学、郑州大学、中国科学技术大学、济南大学、哈尔滨工业大学、浙江大学、香港科技大学、山西大学、郑州轻工业大学、成都信息工程大学、东华大学、成都信息工程大学、复旦大学、东北林业大学、辽东学院、上海电子信息职业技术学院、西南交通大学、南京邮电大学、华中师范大学、临沂职业学院、北京科技大学、东北大学、哈尔滨工业大学（威海）、青海大学、中南大学、北京交通大学、中国矿业大学、玉林师范学院、太原理工大学、芜湖联合大学、广东工业大学、福建工程学院、福建师范大学、北京林业大学、三峡大学、五角场文理学院、山东师范大学、郑州西亚斯学院、南京大学、首都师范大学、黑龙江大学、西安理工大学、北京航空航天大学、许昌学院、河池学院、中国地质大学（北京）、杭州职业技术学院、河海大学、四川师范大学、安徽大学、湖南中医药大学、湘潭大学、济南大学、浙江传媒学院、南辛庄防疫大学、苏州大学、西安交通大学、湖北科技职业学院、华中农业大学、上海财经大学、华侨大学、北方工业大学、长沙理工大学、重庆师范大学、河南工程学院、华南理工大学、蚌埠芜湖大学、暨南大学、广东财经大学、大连理工大学、西安石油大学、四川大学、华南师范大学、哈尔滨工业大学、集美大学、南昌航空大学科技学院、武汉理工大学、应急管理大学、天津工业、东南大学、广东外语外贸大学、浙江工业、武汉科技大学、河北农业大学、北京工业大学、西南民族大学、上海科技大学、燕山大学、河北大学、南京航空航天大学、厦门大学、西南石油大学（成都校区）、山东科技大学、合肥工业大学、齐鲁工业大学（山东省科学院）、辽宁科技大学。

以上大学，虽未开设任何课程，但绝大部分都已经知道 Rust 语言的存在。只有一例，将 Rust 语言误以为是 R 语言。

如何开始学习 Rust 语言?

圆桌收录 Rust 语言圆桌年话 >

以前学 C 有很多书籍，所以没有这种学新语言的经验。请给个简单的路线、链接、或资料。

关注问题 | 写回答 | 邀请回答 | 好问题 30 | 添加评论 | 分享 | ··· | 收起

查看全部 39 个回答

rust永垂不朽

《rust 编程艺术》作者

+ 关注

17 人赞同了该回答

很简单，在公共场合，例如公司，咖啡店等地方写 rust 代码。

如果连续三次编译出错，那么你已经被 rust 感化营的人盯上了。

但对你来说是好事，他们会手把手教你写 rust，等于你白嫖了一次培训。

编辑于 2023-03-23 21:46 · IP 属地广东

▲ 赞同 17 ▼ | 添加评论 | 分享 | 收藏 | 喜欢 | ···

第三步：

- 到了这一步才真正开始 Rust 的学习，关于任何编程语言的学习，都是从阅读官方文档开始。Rust 社区发行了一个简明的开源教程：The Rust Programming Language，强烈推荐阅读。
- 如果不太适应英文教材的阅读，可以看这本书的中文翻译版本：Rust 程序设计语言（第二版 & 2018 edition）简体中文版。
- 在阅读官方文档的时候，就可以根据上述的进行一些编程练习了。不过 Rust 的官方文档相对于其他编程语言文档来说，比较注重的是 Rust 的几个编程特性：零成本抽象，所有权特性等，在阅读这些的时候不推荐自己瞎写练习，应该对比阅读文档和其他的开源 Rust 代码。

第四步：

- 在充分感受 Rust 的编程范式以后，应该进入一定量的练习阶段。
- 这个时候推荐学习的材料是：Rust by Example, Rust by Practice, 直接通过在线修改代码阅读样例来学习。
- 快速过一遍 Rust by Example 之后，相信现在至少可以开始上手写一些 Rust 代码了，个人喜欢把以前的一些小项目用新学习的编程语言重写，例如实现一个最简单的 DBMS，通过这样的行为不仅可以加深对新学语言的熟练度，还可以通过与原有实现的对比，感受 Rust 语言特性。

https://www.zhihu.com/question/31038569/answer/2950409237

上节课回顾

内存管理和安全性

1. 程序和内存
2. 程序如何使用内存
3. 内存管理及其分类
4. 内存分配简介
5. 内存管理的缺陷
6. 内存安全性
7. 内存安全三原则
   （所有权、借用和生命周期）

7. 内存安全三原则（所有权、借用和生命周期）

- Copy 与 Move
- Copy 与 Clone
- 所有权（Ownership）的应用（各类函数调用时所有权的改变）
- Borrow及其规则
- 生命周期

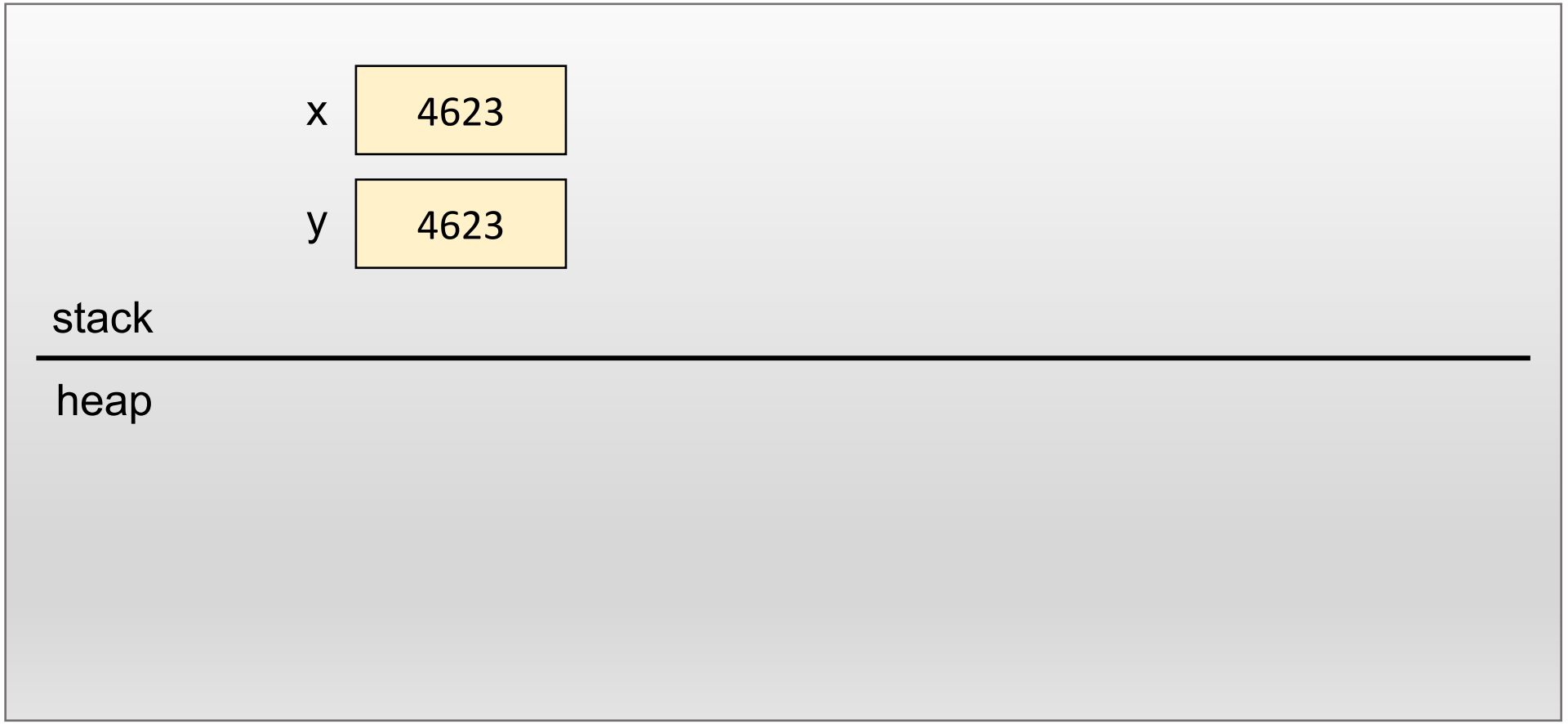
7. 内存安全三原则（所有权、借用和生命周期）

■ Copy 与 Move

```rust
fn main() {
    // copy operation
    let x = 4623;
    let y = x;
    println!("{:?} {:?}", x, y);
    // move operation
    let s1 = String::from("hello");
    let s2 = s1;
    println!("{:?} {:?}", s2, s1); //error
}
```

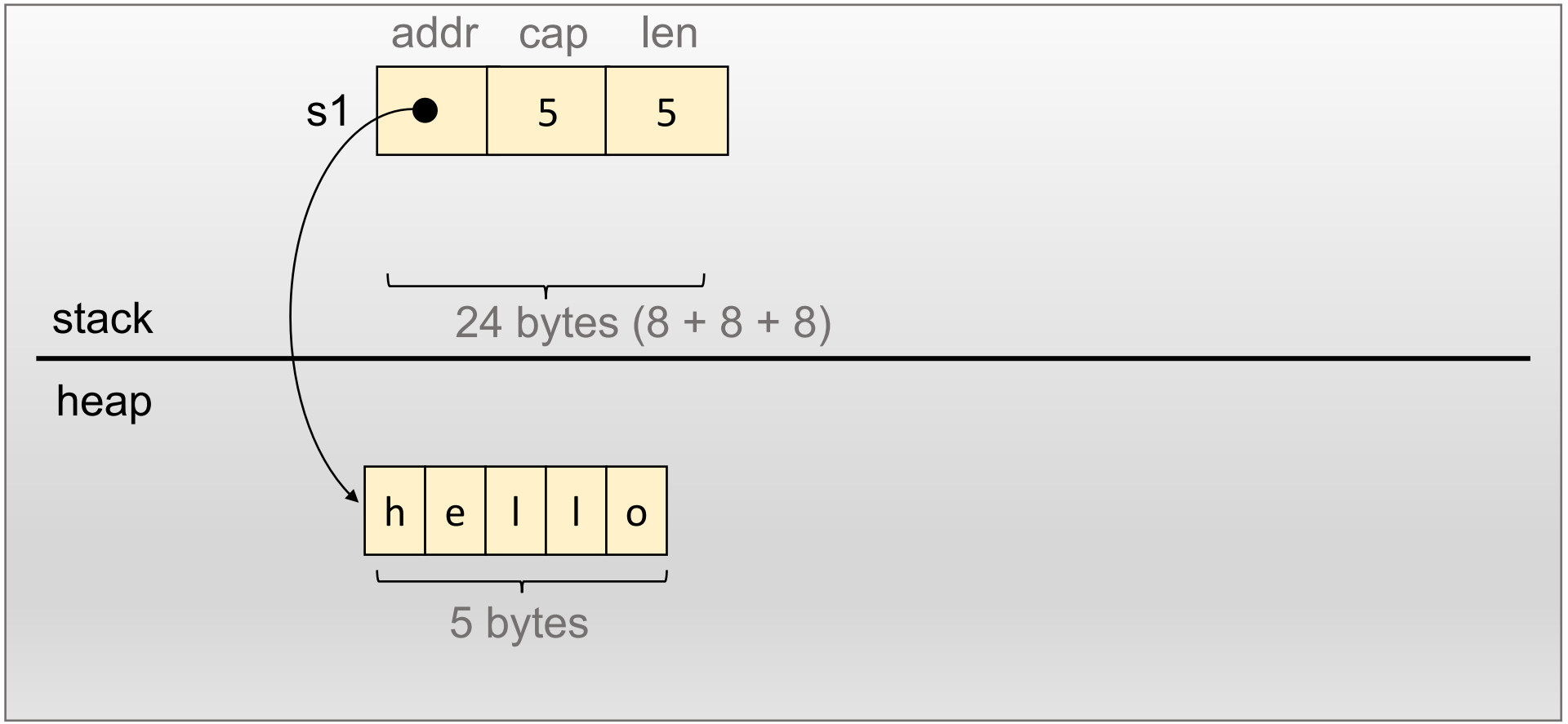
7. 内存安全三原则（所有权、借用和生命周期）

■ Copy 与 Move



7. 内存安全三原则（所有权、借用和生命周期）

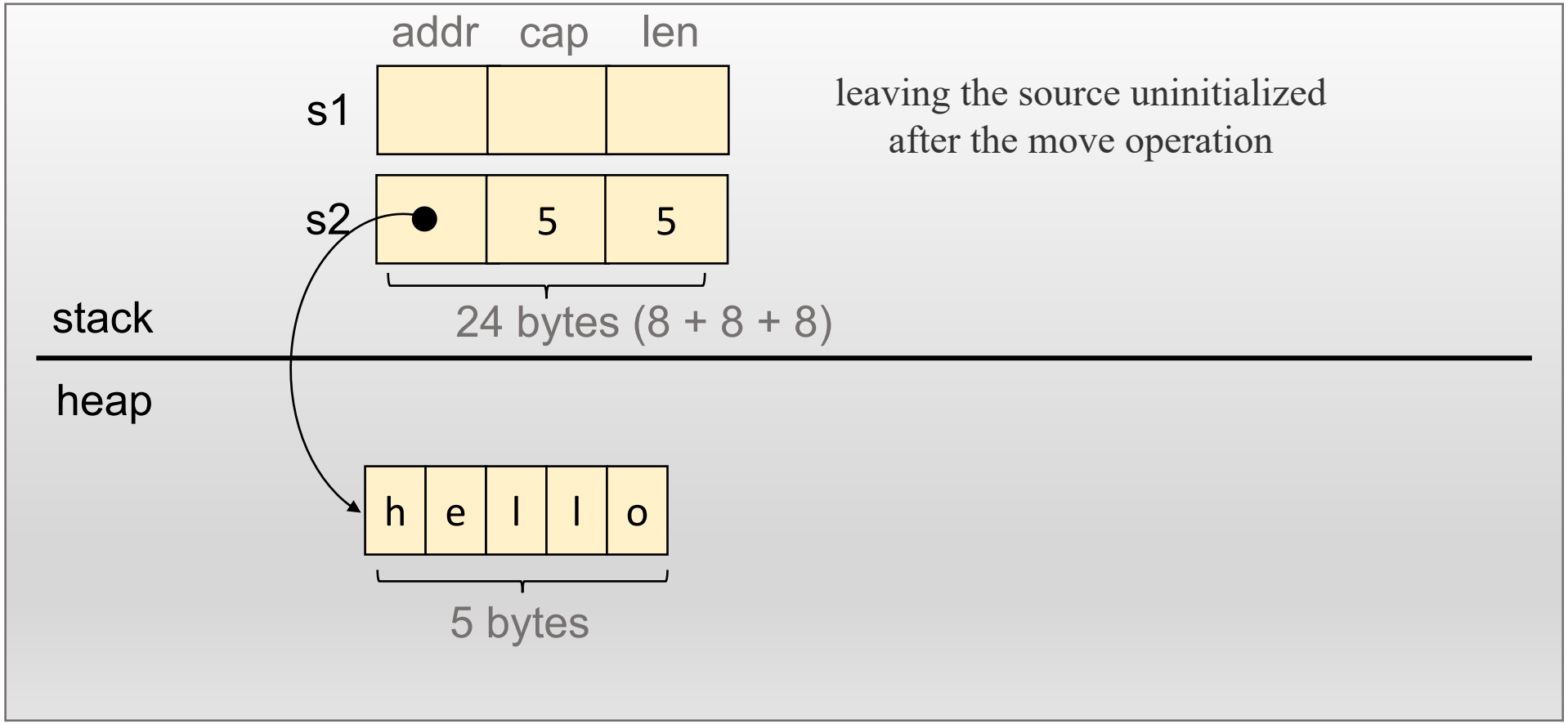
■ Copy 与 Move



More Cases: https://www.oreilly.com/library/view/programming-rust-2nd/9781492052586/ch04.html

7. 内存安全三原则（所有权、借用和生命周期）

■ Copy 与 Move



More Cases: https://www.oreilly.com/library/view/programming-rust-2nd/9781492052586/ch04.html

7. 内存安全三原则（所有权、借用和生命周期）

■ Copy 与 Move

```
fn main() {
    // copy operation
    let x = 4623;
    let y = x;
    println!("{:?} {:?}", x, y);
    // move operation
    let s1 = String::from("hello");
    let s2 = s1;
    println!("{:?} {:?}", s2, s1); //error
}
```

Rust prudently prohibits using uninitialized values, so the compiler rejects this code with the following error:

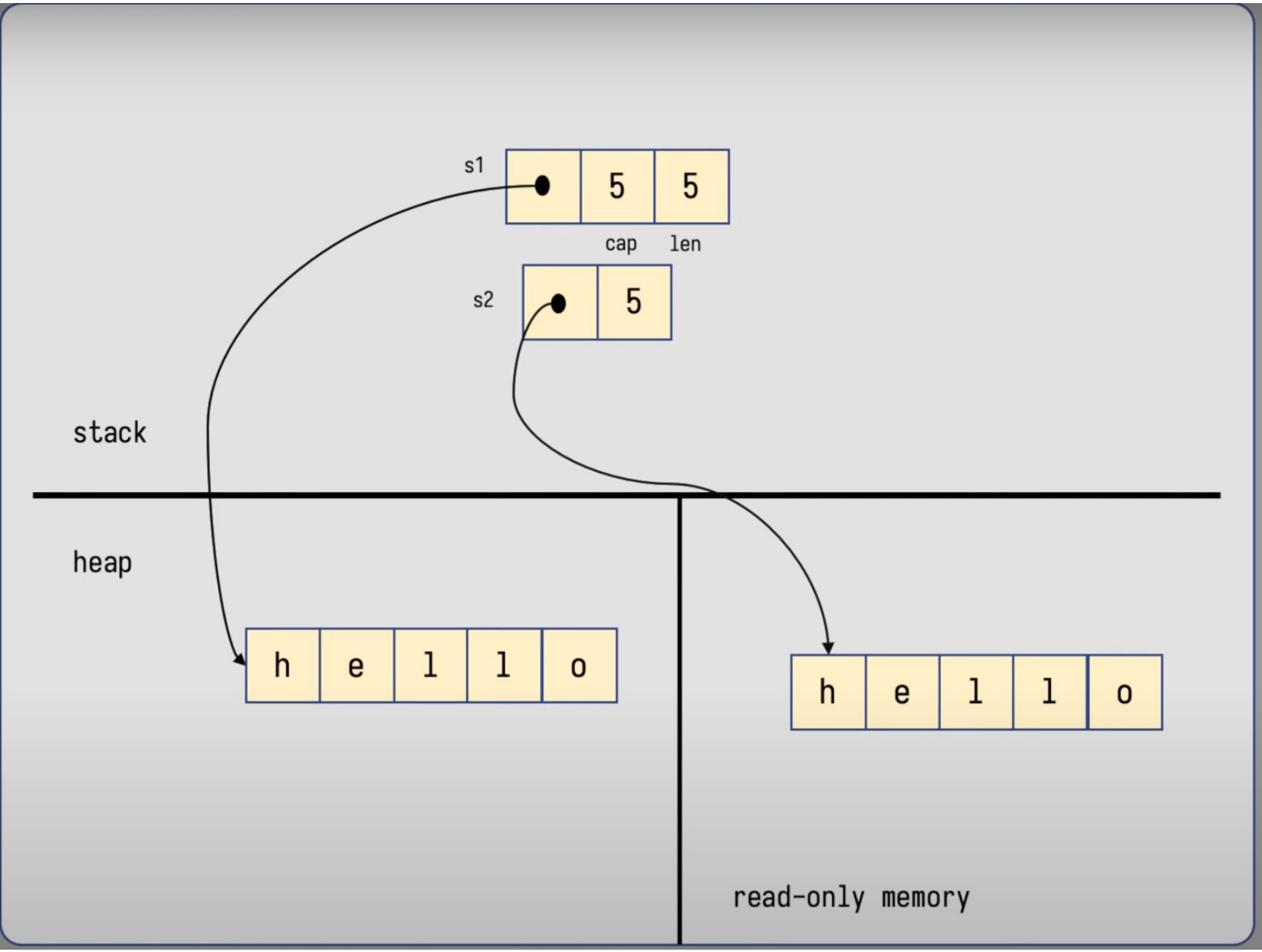
```
let s2 = s1;
         -- value moved here
println!("{:?} {:?}", s2, s1); //error
                          ^^ value borrowed here after move
```

顺便讲解Rust字符串

- 查看变量存储在stack和heap上的开销；

```rust
use std::mem::{size_of, size_of_val};
fn main() {
    let s1 = String::from("hello");
    let s2 = "slice";
    let v = vec![1,2,3,4];
    println!("size of String {}", size_of::<String>());
    println!("size of &str: {}", size_of::<&str>());
}
```

以上会打印出在stack上存储的开销



String

str

&str

```
let s1: String = String::from("hello");
```

```
let s2: &str  = "hello";
```

- **String和 &str（字符串切片）**

顺便讲解Rust字符串

- 查看变量存储在stack和heap上的开销；

```rust
use std::mem::{size_of, size_of_val};
fn main() {
    let s1 = String::from("hello");
    let s2 = "slice";
    let v = vec![1,2,3,4];
    println!("size of String {}", size_of::<String>());
    println!("size of &str: {}", size_of::<&str>());

    println!("size of String: {}, from: {}", size_of_val(&s1), s1);
    println!("size of &str: {}, from: {}", size_of_val(&s2), s2);
}
```

以上会打印出在stack上存储的开销
(另一种方法)

顺便讲解Rust字符串与char类型

- 查看变量存储在stack和heap上的开销；

```rust
use std::mem::{size_of, size_of_val};
use get_size::GetSize;
fn main() {
    let s1 = String::from("hello");
    let s2 = "slice";
    println!("size of String {}", size_of::<String>());
    println!("size of &str: {}", size_of::<&str>());

    println!("size of String: {}, from: {}", size_of_val(&s1), s1);
    println!("size of &str: {}, from: {}", size_of_val(&s2), s2);

    println!("heap size of s1: {}", s1.get_heap_size());
    println!("heap size of s2: {}", s2.get_heap_size());
}
```

使用get-size包，在Cargo.toml文件中添加：

get-size = { version = "^0.1", features = ["derive"] }

顺便讲解Rust字符串与char类型

- 查看变量存储在stack和heap上的开销；

```rust
use std::mem::{size_of, size_of_val};
use get_size::GetSize;
fn main() {
    let s1 = String::from("hello");
    let s2 = "💖💖💖💖💖";
    println!("size of String {}", size_of::<String>());
    println!("size of &str: {}", size_of::<&str>());

    println!("size of String: {}, from: {}", size_of_val(&s1), s1);
    println!("size of &str: {}, from: {}", size_of_val(&s2), s2);

    println!("heap size of s1: {}", s1.get_heap_size());
    println!("heap size of s2: {}", s2.get_heap_size());
}
```

non-ASCII strings的大小又不一样
https://doc.rust-lang.org/std/string/struct.String.html

顺便讲解Rust字符串与char类型

■ 查看变量存储在stack和heap上的开销；

- 在Rust中，一个字符（char）是一个单一的Unicode码点，需要32位来容纳所有这些码点。
- Rust中的字符串以UTF-8编码存储为字节数组，这意味着字符通常只占用一个字节。除非它们超出ASCII范围，此时它们最多可以占用4个字节。
- 而C语言则忽略了这一点，其中一个字符（char）是一个字节，因为C语言不关心你的字符串是如何编码的，它只使用字节来表示字符串，并把编码的管理留给你自己来处理。

7. 内存安全三原则（所有权、借用和生命周期）

■ Borrow及其规则（由于所有权的限制非常严格，我们需要更灵活的操作）

- Immutable Borrow;
- Mutable Borrow;

```
#[derive(Debug)]
struct Foo;

fn main() {
    let foo = Foo;
    let bar = &foo;
    println!("Foo is {:?}", foo);
    println!("Bar is {:?}", bar);
}
```

```
fn main() {
    let mut a = String::from("Owned
string");
    let a_ref = &mut a;
    a_ref.push('!');
}
```

- 可变借用的变量（a）需要添加mut关键词;
- 借用的引用变量（a_ref）需要 &mut 关键词;
- Q: Stack和Heap内存中如何存储 a_ref ?

7. 内存安全三原则（所有权、借用和生命周期）

- Borrow及其规则（由于所有权的限制非常严格，我们需要更灵活的操作）

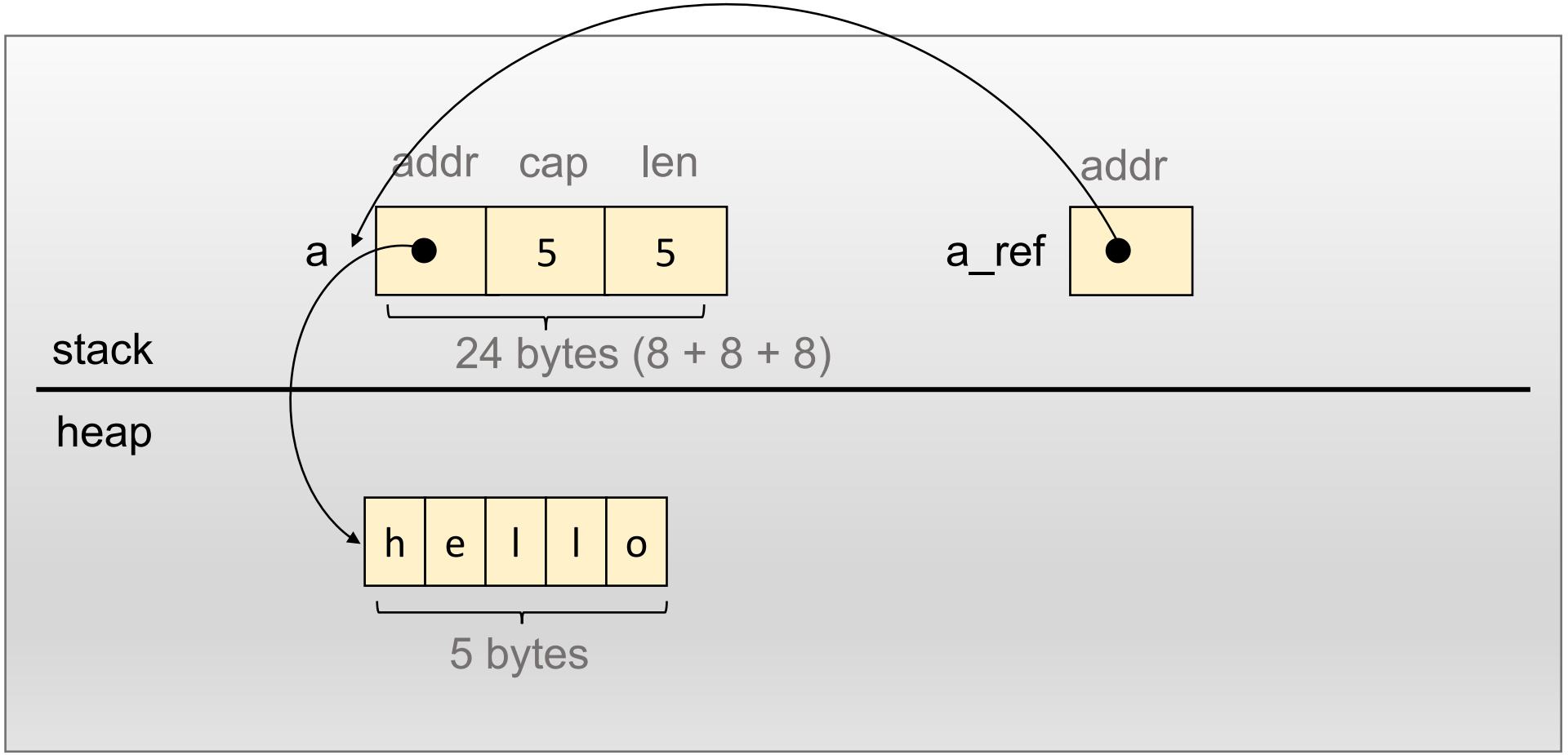
```
let a = String::from(”hello");
let a_ref = &a;

println!("size of a_ref: {}", size_of_val(&a_ref));
println!("size of a: {}", size_of_val(&a));
```

```
     Running `target/debug/hello`
size of a_ref: 8
size of a: 24
```

7. 内存安全三原则（所有权、借用和生命周期）

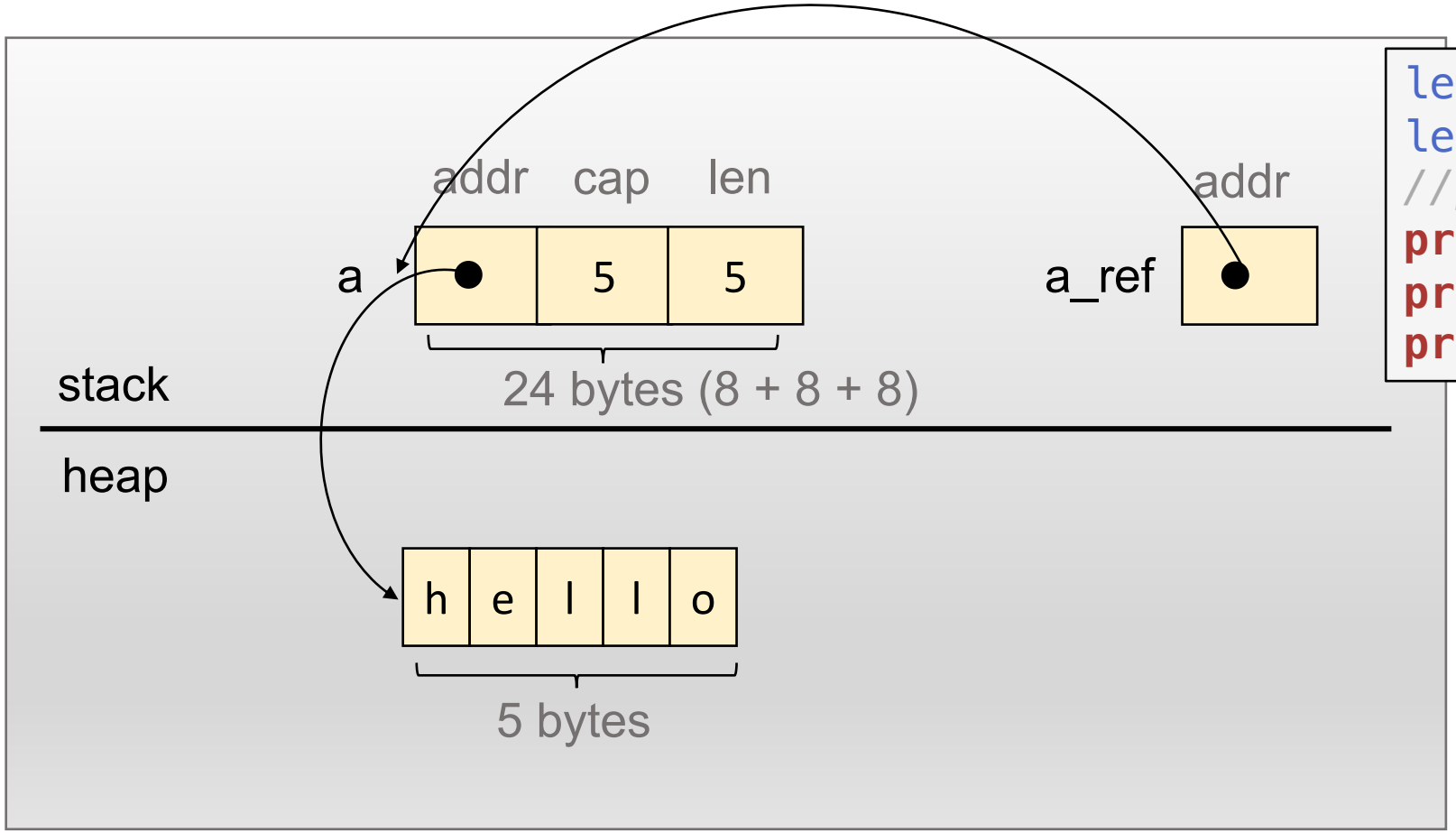
- Borrow及其规则（由于所有权的限制非常严格，我们需要更灵活的操作）



More Cases: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.htmll

7. 内存安全三原则（所有权、借用和生命周期）

■ Borrow及其规则（由于所有权的限制非常严格，我们需要更灵活的操作）




```rust
let a = String::from("hello");
let a_ref = &a;
//print pointer address
println!("{:p}", a.as_ptr());
println!("{:p}", a_ref);
println!("{:p}", &a_ref);
```

```
0x7faa79700040
0x7ffee48b6d40
0x7ffee48b6d58
```

println!默认只显示指针指向的对象，可通过添加{:p}来打印指针本身：https://stackoverflow.com/questions/27852613/why-does-printing-a-pointer-print-the-same-thing-as-printing-the-dereferenced-po

as_ptr方法: https://doc.rust-lang.org/std/primitive.str.html#method.as_ptr

第一次小作业

通过编写一个程序，帮助生成一些术语，将长名称（例如Portable Network Graphics）转换为其首字母缩写（PNG）。

标点符号的处理方式如下：破折号视为单词分隔符（类似于空格）；其他所有标点符号都可以从输入中删除。

For example:

Input	Output
As Soon As Possible	ASAP
Liquid-crystal display	LCD
Thank George It's Friday!	TGIF

考核点：字符串、流程控制、函数

```
fn acronym(input: &str) -> String {

}
```

第一次小作业

通过编写一个程序，帮助生成一些术语，将长名称（例如Portable Network Graphics）转换为其首字母缩写（PNG）。

标点符号的处理方式如下：破折号视为单词分隔符（类似于空格）；其他所有标点符号都可以从输入中删除。

For example:

Input	Output
As Soon As Possible	ASAP
Liquid-crystal display	LCD
Thank George It's Friday!	TGIF

考核点：字符串、流程控制、函数
提交：lib.rs文件包含以下函数：

```
fn acronym(input: &str) -> String {

}
```

第一次小作业

考核点：字符串、流程控制、函数
提交：lib.rs文件包含以下函数：

```
fn acronym(input: &str) -> String {

}
```

2023秋rust课程第一次作业收集
截止时间：2023-10-06 23:59
提交地址：https://send2me.cn/bOWE7K8K/QtuGQ72gRHbHzw

我可以用 GitHub Copilot、ChatGPT 等人工智能的方法进行代码补全完成作业吗?

不可以。

我可以用 ChatGPT 等人工智能的方法帮助学习和理解作业吗?

可以，核心是不能抄代码。例如你可以询问 ChatGPT 一些关于实现思路的问题，但是不要让 ChatGPT 直接生成代码，人性是经不住考验的，所以不要考验自己。如果你觉得自己不能拒绝诱惑，那就不要用 ChatGPT。

如果你使用 ChatGPT 辅助学习，同时也要记住 ChatGPT 提供的答案不一定是正确的。如果 ChatGPT 的回答有所帮助，可以写在代码注释或者 HONOR CODE 中，格式：

在代码中：

```
// Inspired from ChatGPT:
// <answer from ChatGPT>
```

新的内容：类型、泛型（generic）和特征（trait）

面向对象的Rust

类型、泛型（generic）和特征（trait）

1. 类型系统
2. 泛型
3. 特征

1. 类型系统

■ 为什么我们需要类型?

- 类型系统为我们开发人员很好地抽象了内存存储，使我们不必考虑这些令人困惑的0和1。
- 可以将注意力集中在更重要的问题上。

1. 类型系统

■ Why do we need types ?

- 类型系统为我们开发人员很好地抽象了内存存储，使我们不必考虑这些令人困惑的0和1。
- 可以将注意力集中在更重要的问题上。

```
let x: i32 = 1107296256;
let y: f32 = 32.0;
let ybit = y.to_bits();
println!("The binary code of x is: {:b}", x);
println!("The binary code of ybit is: {:b}", ybit);
```

1. 类型系统

■ Why do we need types ?

- 类型系统为我们开发人员很好地抽象了内存存储，使我们不必考虑这些令人困惑的0和1。
- 可以将注意力集中在更重要的问题上。

```
let x: i32 = 1107296256;
let y: f32 = 32.0;
let ybit = y.to_bits();
println!("The binary code of x is: {:b}", x);
println!("The binary code of ybit is: {:b}", ybit);
```

```
    Running `target/debug/hello_world`
The binary code of x is: 1000010000000000000000000000000
The binary code of ybit is: 1000010000000000000000000000000
```

1. 类型系统

■ Rust中的类型系统

- Haskell是一种高级语言，具有非常丰富的、有表现力的类型系统；
- C语言是一种低级语言，只提供了很少的基于类型的抽象。
- Rust寻求了一种平衡。

Rust借鉴了Haskell等函数式语言的特点，如抽象数据类型中的结构体、特征（类似于Haskell的类型类）、及错误处理类型（Option和Result）。

1. 类型系统

■ Rust中的类型系统

- Haskell是一种高级语言，具有非常丰富的、有表现力的类型系统；
- C语言是一种低级语言，只提供了很少的基于类型的抽象。
- Rust寻求了一种平衡。

Haskell is both statically and dynamically typed language where the compiler infers the types of any variables that are written and the value assigned cannot be changed at the runtime.	Rust is a statically typed language and therefore it identifies the error at runtime which is very easy for debugging when compared to other dynamically typed language.

https://www.educba.com/haskell-vs-rust/

2. 泛型（generic）

- **目的：追求更好的抽象，追求代码复用**
  - 例如，编写一个函数，接收不同类型的输入。

- **泛型编程是仅适用于静态类型编程语言的技术**
  - 像Python这样的动态语言采用的是简单类型（duck typing）

    If it walks like a duck, and it quacks like a duck, then it must be a duck.

  - Rust的泛型系统在编译时（compile time）决定变量类型。

2. 泛型（generic）

- **目的：追求更好的抽象，追求代码复用**
  - 例如，编写一个函数，接收不同类型的输入。

- **泛型编程是仅适用于静态类型编程语言的技术**
  - 像Python这样的动态语言采用的是简单类型（duck typing）

    If it walks like a duck, and it quacks like a duck, then it must be a duck.

  - Rust的泛型系统在编译时（compile time）决定变量类型。
  - 零成本抽象（Zero Cost Abstraction）

2. 泛型（generic）

- **标准库中的泛型举例**
  - **Vec<T>**

- **泛型编程是仅适用于静态类型编程语言的技术**
  - 像Python这样的动态语言采用的是简单类型（duck typing）
  - Rust的泛型系统在编译时（compile time）决定变量类型。

2. 泛型（generic）

■ 2.1 创建泛型——创建一个接收不同类型输入的函数

```
fn give_me<T>(value: T) {
    let _ = value;
}

fn main() {
    let a = "generics";
    let b = 1024;
    give_me(a);
    give_me(b);
}
```

要点：

- 输入变量名后的类型变为T；
- 函数名后面添加<T>；
- T可以被替换成其他字母或者字符串（CamelCase命名法）；

2. 泛型（generic）

■ 2.1 创建泛型——创建一个接收不同类型输入的函数

要点：

- 你编写的代码不是最终的代码，而是一种模板，泛型的类型参数（T）是一种占位符。
- 终端输入“nm target/debug/hello_world|grep give”，信息如下：

```
0000000100001370 t __ZN11hello_world7give_me17h05aeb6c93bcf5019E
0000000100001390 t __ZN11hello_world7give_me17hc9e3a3893091eedeE
```

- 泛型提供了一种多态代码的错觉，所谓错觉是指编译后的文件实际是包含具体类型参数的重复代码。（请思考：优缺点分别是什么？）

2. 泛型（generic）

■ 2.1 创建泛型——创建一个泛型结构体

```
struct GenericStruct<T>(T);


struct Container<T> {
    item: T
}
```

- GenericStruct为元组结构体；
- Container为一般结构体（键/值对形式）。

2. 泛型（generic）

■ 2.1 创建泛型——创建一个泛型枚举（Enum）

```
enum Transmission<T> {
    Signal(T),
    NoSignal
}
```

2. 泛型（generic）

■ 2.3 泛型应用——Vec<T>的使用

```
let a = Vec::new();
```

能否编译通过?

2. 泛型（generic）

■ 2.3 泛型应用——Vec<T>的使用

```
fn main() {
    // providing a type
    let v1: Vec<u8> = Vec::new();
    // or calling method
    let mut v2 = Vec::new();
    // v2 is now Vec<i32>
    v2.push(2);
    // or using turbofish
    let v3 = Vec::<u8>::new();    // not so readable
}
```

要点：

- 明确指明变量类型；
- 调用方法让编译器推断类型；
- 使用turbofish方法，注意错误的案例为： `let v3 = Vec<u8>::new();`

3. 特征（trait）

- 从多态和代码复用的角度来看，将类型的共享行为和公共属性与其自身隔离通常是一个好主意。
- 类似Java和C#的面向对象编程中的接口，可以为多种类型实现共享的行为。
- **Rust中类似且功能强大的结构，被称为特征（trait）。**

面向对象的Rust

- Rust中类似且功能强大的结构，被称为特征（trait）。

面向对象的Rust

- 为结构化数据类型（struct、enum）实现方法

为结构化数据类型（struct、enum）实现方法

```rust
impl Point {
    pub fn distance(&self, other: Point) -> f32 {
        let (dx, dy) = (self.x - other.x, self.y - other.y);
        ((dx.pow(2) + dy.pow(2)) as f32).sqrt()
    }
}
fn main() {
    let p = Point { x: 1, y: 2 };
    p.distance();
}
```

- 结构体和枚举的方法可以实现在 `impl` 代码块里。
- 和域相同，方法也通过点记号进行访问。
- 可以用 `pub` 将方法声明为公开的，`impl` 代码块本身不需要是 `pub` 的。
- 对枚举和对结构体是一样的。

方法与所有权

方法的第一个参数（名字为 `self`）决定这个方法需要的所有权种类。

- `&self`：方法**借用**对象的值。
  - 一般情况下尽量使用这种方式，类似于 C++ 中的常成员函数。
- `&mut self`：方法**可变地借用**对象的值。
  - 在方法需要修改对象时使用，类似于 C++ 中的普通成员函数。
- `self`：方法获得对象的所有权。
  - 方法会消耗掉对象，同时可以返回其他的值。

关联函数

```rust
impl Point {
    fn new(x: i32, y: i32) -> Point {
        Point { x: x, y: y }
    }
}

fn main() {
    let p = Point::new(1, 2);
}
```

- 关联函数与方法类似，但是没有 `self` 参数。
  - 调用时使用名字空间语法：`Point::new()`，而不是 `Point.new()`。
  - 类似 C++ 中的静态成员函数。
- 一般会创建一个名为 `new` 的关联函数起到构造函数的作用。
  - Rust 没有内置的构造函数语法，也不会自动构造。

Object Oriented Programming in C++

Classes

- "面向对象"：创建一个'对象' - 电影数据库，并可以对该对象执行方法。
- 您可以创建对象的实例，每个实例都有自己的变量集合（具有不同的文件的电影数据库）。
- 类分为公共和私有区域。
- 公共成员可供任何具有对实例的引用的人访问。 私有成员仅可由类的实现者访问。

```
class imdb {
  public:
    imdb(const std::string& directory)
    bool getCredits(...)
  private:
    /* Elements
    const char* kActorFileName;
}
```

What are some advantages to Classes?

Advantages to Class Design

- 模块化：我们可以将一个大型系统分解为可管理的组件，提供清晰的接口，并可以独立测试。
- 封装性：将相关的数据和方法组合到一个单一的“对象”中。
- 隐藏代码：不需要暴露给用户与之交互所需的类的不必要部分。
- 代码重用：想要一个对象基于它接收的文件而有所不同吗？只需向其构造函数添加一个参数，突然间你就有了两种不同的实现，但只有一个类！

Reusing code with “inheritance”

A bunch of slightly different types of teddy

```
class TeddyBear {
  public:
    TeddyBear(..);
    void roar_sound();
}
```

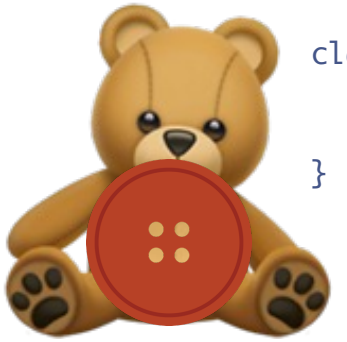
```
class PurpleTeddyBear {
  public:
    TeddyBear(..);
    void roar_sound();
    void purple_button_song();
}
```

```
class RedTeddyBear {
  public:
    TeddyBear(..);
    void roar_sound();
    void red_button_song();
}
```

```
class PurpleTeddyBear {
  public:
    TeddyBear(..);
    void roar_sound();
    void green_button_song();
}
```

Inheritance

```
class TeddyBear {
   public:
      TeddyBear(..);
      void roar_sound();
}
```



```
class RedTeddyBear {
   public:
      red_button_song();
}
```

```
class PurpleTeddyBear {
   public:
      purple_button_song();
}
```

```
class GreenTeddyBear {
   public:
      green_teddy_bear();
}
```

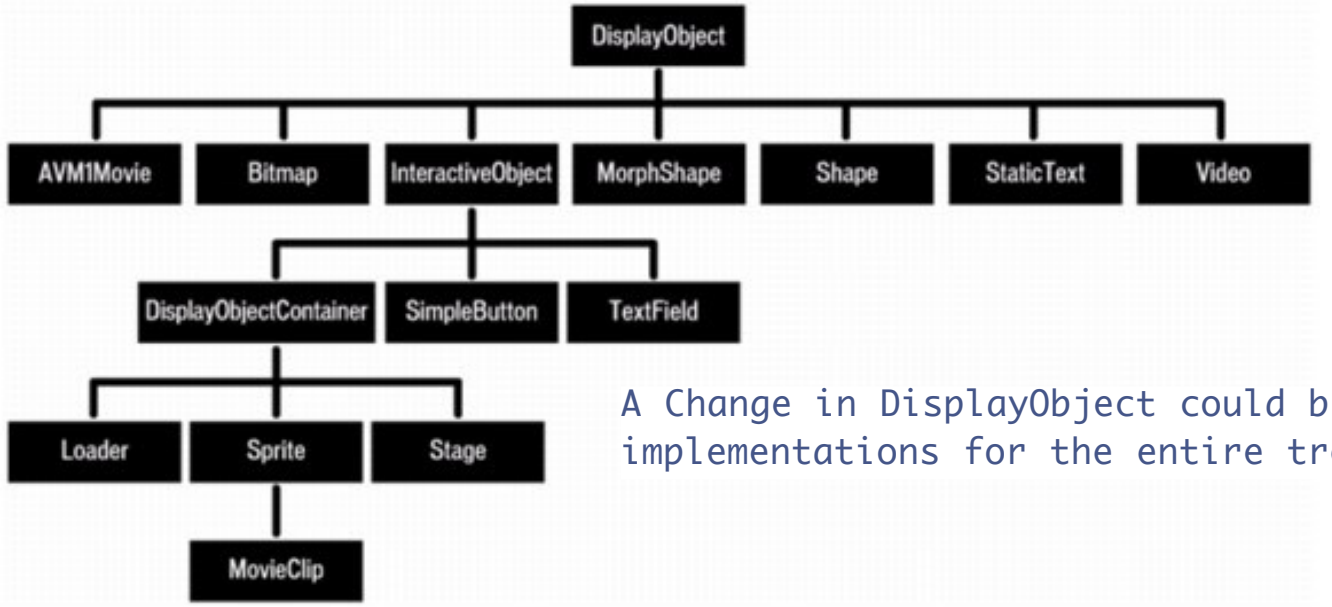
Lets take a look!

继承（ Inheritance ）

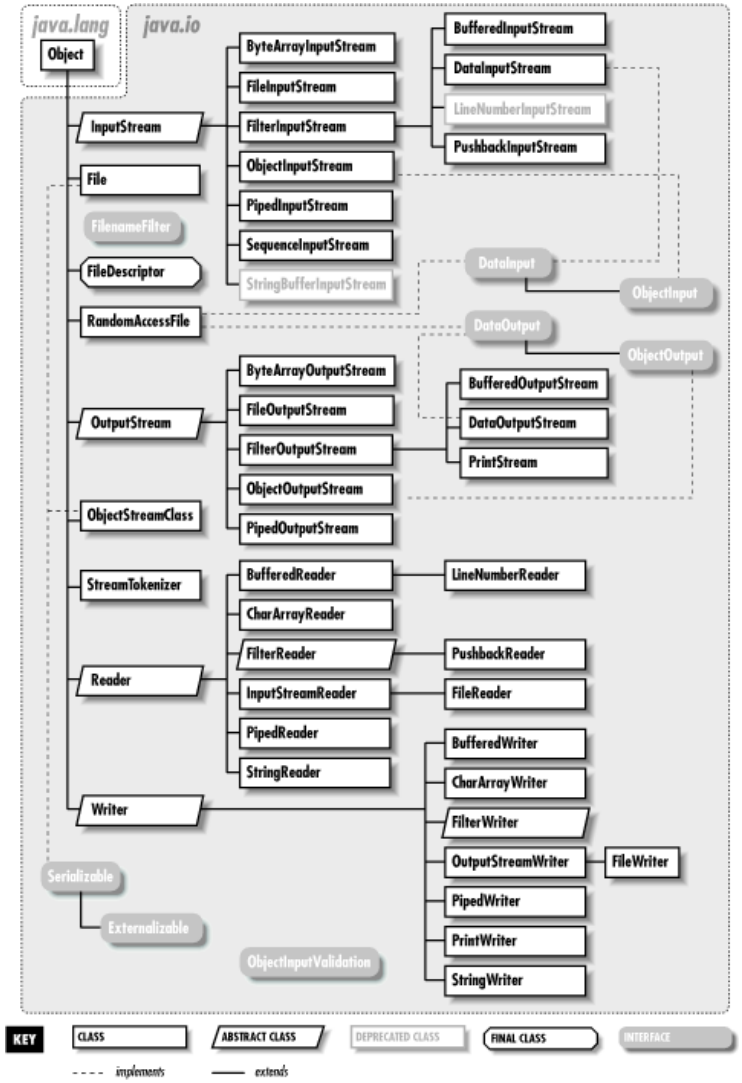
- 通过继承，我们能够在许多不同类型的对象中使用相同的方法实现，通过父子关系将它们组合在一起。
- 子类继承所有的方法和属性（通常不包括构造函数，具体取决于编程语言）。它们可以选择覆盖父类函数（例如，绿熊发出不同的咆哮声）。
- 在像Java这样的语言中，继承是一个重要的概念（在那里，几乎所有的类都继承自一个基础的Object类）。

What might be the weaknesses of Inheritance?

Inheritance Trees



A Change in DisplayObject could break implementations for the entire tree!



思考：随着时间的推移，如何维护和更改一个庞大的代码库。

Aside: Two Other Keywords

- 对象组合
  - 类A具有其他类类型的实例变量。
  - 例如：想要制造多种类型的填充动物。定义诸如“毛发”、“羽毛”、“爪子”、“嘴巴”等，并将它们组合在一起以创建更复杂的填充动物。
  - 松散的耦合：如果可能的话，通常是比继承更好的选择。

- 多态性
  - 不同的底层类型/实现共享一个单一的接口
  - 例如：绿熊从（基础）熊继承了“咆哮”，但绿熊的“咆哮”实现方式不同。

特征（Traits）

我们还可以以哪些其他方式进行分解？

https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=da8b2ac99e2c386656cb103c277a014e

```
struct TeddyBear;

impl TeddyBear {
  fn roar(&self) {
    println!("ROAR!!");
  }
}
```

```
struct PurpleTeddyBear;

impl PurpleTeddyBear {
  fn roar(&self) {
    println!("ROAR!!");
  }
  fn purple_button_song(&self){
    /* Purple Song */
  }
}
```

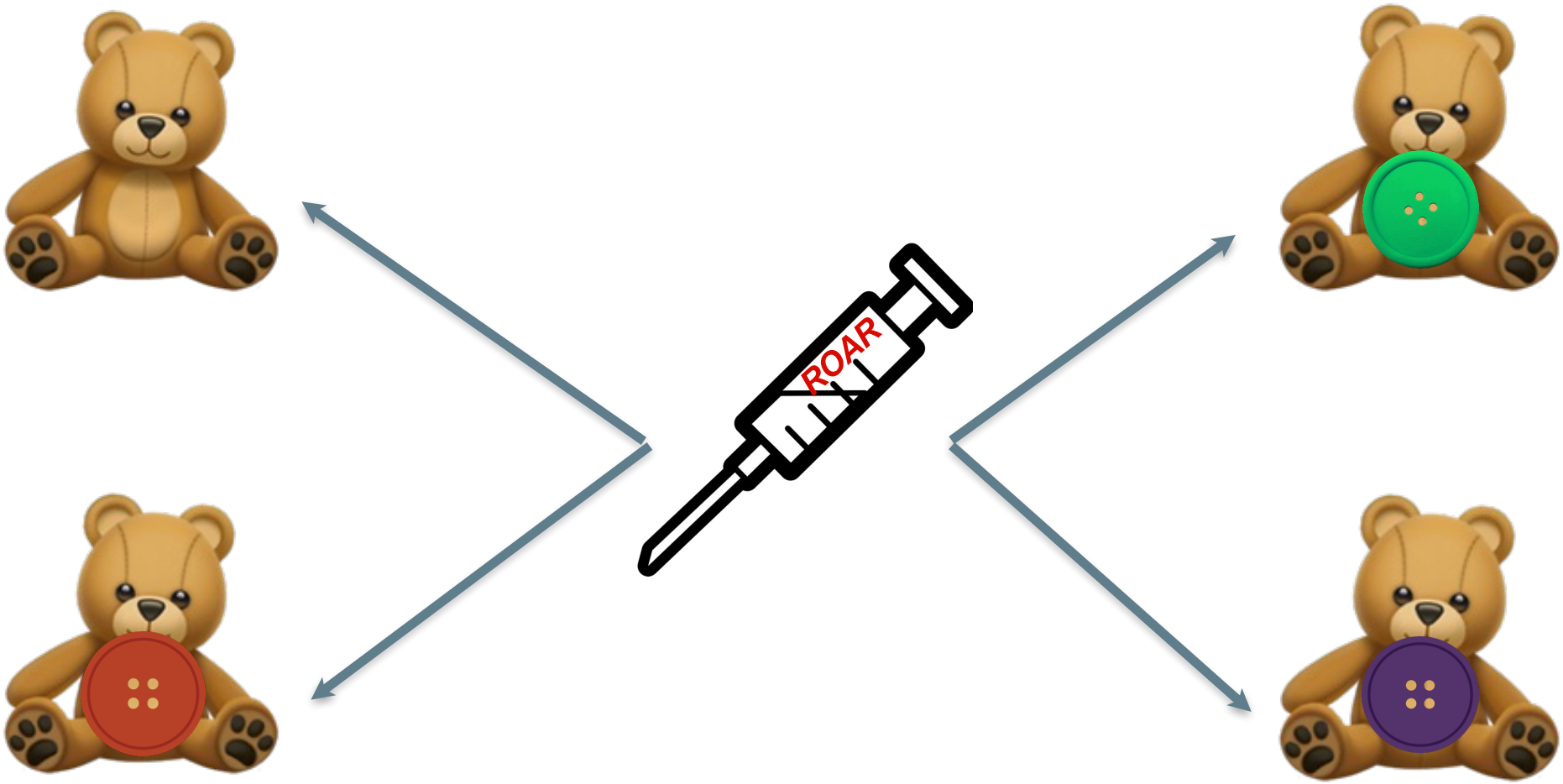
```
struct RedTeddyBear;

impl RedTeddyBear {
  fn roar(&self) {
    println!("ROAR!!");
  }
  fn red_button_song(&self){
    /* Red Song */
  }
}
```

```
struct GreenTeddyBear;

impl GreenTeddyBear {
  fn roar(&self) {
    println!("ROAR!!");
  }
  fn green_button_song(&self){
    /* Green Song */
  }
}
```

Traits



Inject the code you want into the other classes! (Inject a trait into them!)

Let's make our first trait!

Traits Overview

- 使用特征（traits），您可以编写可以注入到任何现有结构中的代码。（从TeddyBear到i32！）此代码可以引用self，因此代码可以依赖于实例。
- 特征方法不需要完全定义 - 您可以定义一个在为某种类型实现特征时必须实现的函数。（类似于Java接口）
- 特征可以指定函数/数据实例应该具有的内容，而不仅仅是从另一个“父级”获取许多。
- 不再需要深层继承层次结构。只需思考：“这个类型是否实现了这个特质？”

Background, if you're interested:
https://blog.rust-lang.org/2015/05/11/traits.html

Questions?

Rust主要的标准Trait

标准Traits

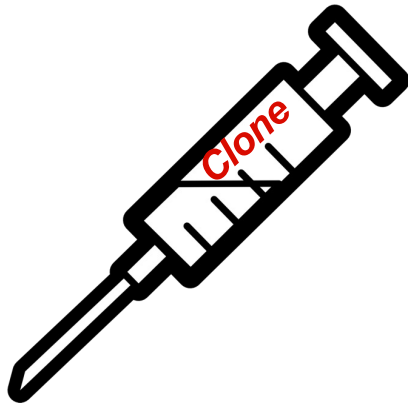
- Copy: 在使用赋值（=）时将创建一个实例的新副本，而不是移动所有权。
- Clone: 在调用方法的.clone()函数时将返回一个实例的新副本。
- Drop: 将定义释放实例内存的方式 - 当实例达到作用域末尾时调用。
- Display: 定义了一种格式化类型和显示它的方式（由println！使用！）
- Debug: 类似于Display，但不是面向用户的（用于调试类型！）
- Eq: 定义了两个相同类型对象的相等性的确定方式（由等价关系定义）。
- PartialOrd: 定义了比较实例的方式（小于、大于、小于或等于等）。

Lets implement a standard Trait!

```
struct Point {
    x: u32,
    y: u32,
}

fn main() {
  let pt = Point {x:3, y:2};
  let pt2 = pt.clone();
}
```

Does not compile - clone() isn't defined



Let's Inject Clone!

https://doc.rust-lang.org/std/clone/trait.Clone.html

Injecting Clone: recap

- 只要它们是兼容的（Drop 与 Copy 不兼容），您可以将任何特质实现到任何结构中，就像我们对 Point 实现 Clone 一样。
- 您可以使用Rust文档作为一种方式，告诉您需要实现哪些函数以及它们的参数类型。
- 您可以使用 #[derive(x, y, z..)] 来派生特质。如果您的结构满足一些规则（由文档给出），Rust编译器将尝试为您实现这些特质。例如：如果结构中的所有成员已经实现了 Clone，您可以派生 Clone 特质。

特征（trait）例子

- **Rust中类似且功能强大的结构，被称为特征（trait）：**
- **从一个案例来看：**
  - 我们要构建一个多媒体播放器（可播放视频和音频），具有播放 (play) 和暂停 (pause) 的功能
  - 如果用结构体 + Impl关键词如何实现？

特征（trait）例子

- **Rust中类似且功能强大的结构，被称为特征（trait）：**
- **从一个案例来看：**
  - 我们要构建一个多媒体播放器（可播放视频和音频），具有播放 (play) 和暂停 (pause) 的功能
  - 如果用结构体 + Impl关键词如何实现？
    - Audio 和 Vedio结构体，分别实现play和pause函数；
    - 共享的功能是使用特征的时机，提升代码复用性。

特征（trait）例子

- Rust中类似且功能强大的结构，被称为特征（trait）：
- 从一个案例来看：

```rust
struct Audio(String);
struct Video(String);

trait Playable {
    fn play(&self);
    fn pause() {
        println!("Paused");
    }
}
```

```rust
impl Playable for Audio {
    fn play(&self) {
        println!("🎵 Now playing: {}", self.0);
    }
}

impl Playable for Video {
    fn play(&self) {
        println!("🎵 Now playing: {}", self.0);
    }
}
```

假设写在同一个文件 main.rs中

特征（trait）例子

- Rust中类似且功能强大的结构，被称为特征（trait）：
- 从一个案例来看（拆分成文件）：

super_player/src/main.rs

```rust
struct Audio(String);
struct Video(String);

impl Playable for Audio {
    fn play(&self) {
        println!("🎵 Now playing: {}", self.0);
    }
}
impl Playable for Video {
    fn play(&self) {
        println!("🎵 Now playing: {}", self.0);
    }
}
```

super_player/src/media.rs

```rust
trait Playable {
    fn play(&self);
    fn pause() {
        println!("Paused");
    }
}
```

测试能否编译通过?

特征（trait）例子

- Rust中类似且功能强大的结构，被称为特征（trait）：
- 从一个案例来看（拆分成文件）：

super_player/src/main.rs

```
mod media;
use media::Playable;

struct Audio(String);
struct Video(String);
impl Playable for Audio {
    fn play(&self) {
        println!("🎵 Now playing: {}", self.0);
    }
}
impl Playable for Video {
    fn play(&self) {
        println!("🎵 Now playing: {}", self.0);
    }
}
```

super_player/src/media.rs

```
pub trait Playable {
    fn play(&self);
    fn pause() {
        println!("Paused");
    }
}
```

特征（trait）例子

- **特征的继承，某个特征依赖其他特征**
- **从另一个案例来看：**
  - 我们要构建一个特斯拉TeslaRoadster对象，具有Vehicle和Car特征。

特征（trait）例子

- **特征的继承，某个特征依赖其他特征**
- **从另一个案例来看：**
  - 我们要构建一个特斯拉TeslaRoadster对象，具有Vehicle和Car特征。

```
trait Vehicle {
    fn get_price(&self) -> u64;
}

trait Car: Vehicle {
    fn model(&self) -> String;
}
```

```
struct TeslaRoadster {
    model: String,
    release_date: u16
}

impl Car for TeslaRoadster {
    fn model(&self) -> String {
        "Tesla Roadster I".to_string()
    }
}
```

Q & A

Thanks!